Hugo BESCH Erasmus student

Tyre Quality Classification

-

Documentation & Blog post

Keras TensorFlow

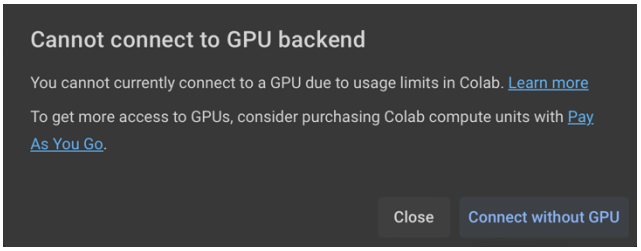
TABLE OF CONTENTS

INTRODUCTION

For the semester project I choose to work on the Tyre Quality Classification project.

I have a particular attraction for machine learning about images feature extractions. I think this can be beneficial for many industries, that's why I choose this subject.

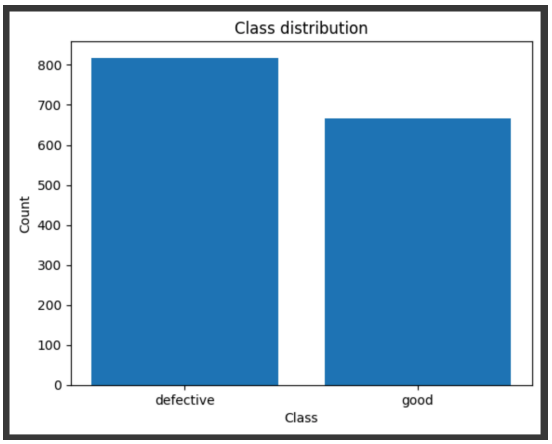
As I'm working on a MacBook and the AMD graphic cards still not compatible with TensorFlow (actually there are some tricks but it's not stable), I was forced to use Google Colab with T4 GPU which can be slow sometimes and kind of a pain.



Sad ☹️

DATA

The dataset contains 1856 digital tyre images, categorized into two classes: defective and good condition. Each image is in a digital format and represents a single tyre. The images are labelled based on their condition whether the tyre is defective or in good condition.



This dataset can be used for various machine learning and computer vision applications, such as image classification and object detection. The automotive industry, and quality control can use the dataset to improve the tyre industry's quality control process and reduce the chances of accidents due to faulty tyres. The availability of this dataset can facilitate the development of more accurate and efficient inspection systems for tyre production.

For the preprocessing step it's important for all the images to have the size so they are so that they can be processed in their entirety and all the information can be found on them. I choose to set which is common in machine learning. I don't think it need rescaling because their isn't a big difference between the quantity of "defective" and "good" tyre in the dataset. Batch size has been set too according to the memory available.

Then training and validation data has been splitted using "tf.keras.utils.image_dataset_from_directory" and specifying previous parameters, seed, validation size…

Classes "defective" and "good" are well detected.

MODELS

THE THEORY

First, it's important to understand how to extract features from an image.

An image is a condensation of data, called pixels. In this first part, we'll take a closer look at the notion of kernel applied to an image.

Convolution is a mathematical operation symbolized by "*" (it is not a simple multiplication often represented as "."). A convolution takes two mathematical objects (as function or matrix) reverse and shift one and calculated the integral of the product.

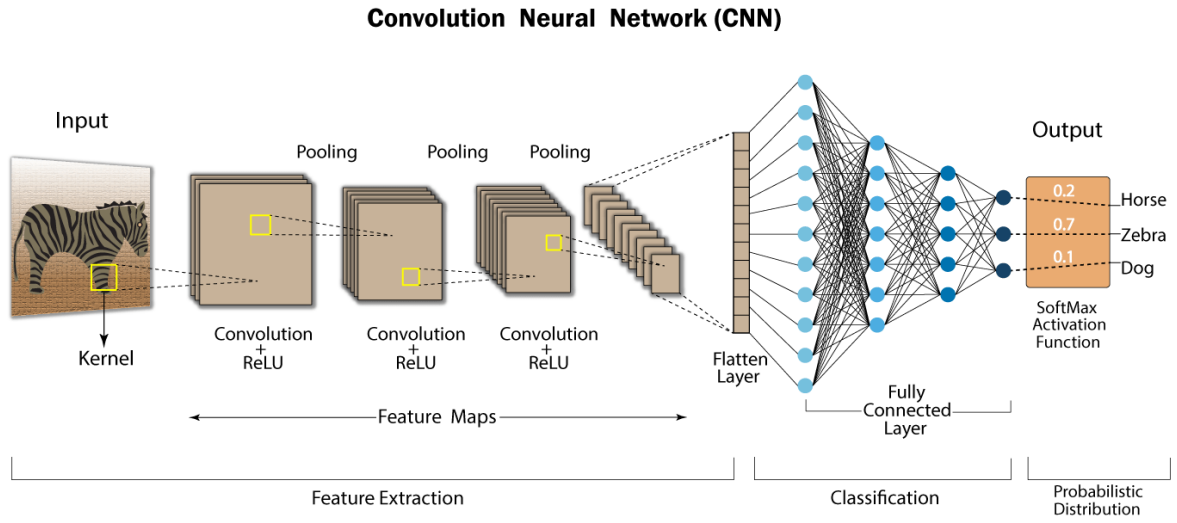
Example:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right)[2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

```
Matrix
  10  10   1
  10   1   1
 -10   1   1
Kernel
  10  10  10
   0   0   0
 -10 -10 -10
Convolution
 110 120  20
-290-290 -90
-110-120 -20
```

We can therefore think of a kernel as a filter applied to the image to extract features. In a CNN, it's the network itself that finds the filters to apply to the images.

The CNN steps are the following:



The input layer receives the raw data.

- Convolutional layer

Apply kernels to the input image. These filters detect features such as edges, textures, or patterns by performing a convolution operation (element-wise multiplication and summation) between the filter and local regions of the input data. Each filter generates a feature map, which highlights the presence of a specific feature in the input.

- Activation Function

After the convolution operation, an activation function like ReLU (Rectified Linear Unit) is typically applied to introduce non-linearity into the network. ReLU sets all negative values to zero, helping the network learn complex patterns.

- Pooling Layer

Pooling layers reduce the spatial dimensions of the convolutional layers' output (feature maps) by downsampling. Max pooling is a common technique where the maximum value from a set of values in a specific window is retained, effectively reducing the size of the feature maps while preserving important information. It reduces the computation complexity at the time.

- Flattening

The output from the previous layers is flattened into a single vector. This process converts the multi-dimensional feature maps into a one-dimensional array to be fed into the subsequent fully connected layers.

- Fully Connected (Dense)

These layers process the flattened features from the previous layers using traditional neural network techniques.

Each neuron in a fully connected layer is connected to every neuron in the previous layer, helping the network learn complex patterns and make predictions.

My work

1st CNN

In my project I first created a CNN model that wasn"t so good. We are going to review this one then compares it to a pretrained and finetuned model named ResNet50. Of course, the pretrained model will reach 98-99% of accuracy. The goal will be to improve my first model to get closer to the results with the pretrained model while avoiding overfitting.

Let"s walk through a Keras "Sequential" object and all the layer in it. This is what contains all the CNN layers.

```
model = tf.keras.Sequential([
    layers.experimental.preprocessing.Rescaling(1./255), # Normalization
    layers.Conv2D(64,3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32,3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(16,3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(64,activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])

model.compile(optimizer='adam', loss=tf.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['accuracy'])

logdir="/content/drive/MyDrive/Tyre Quality Classification/logs"
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir,histogram_freq=1, write_images=logdir, embeddings_data=train_data)

model.fit(train_data,
          validation_data=val_data,
          epochs=10,
          callbacks=[tensorboard_callback]
)
```

The CNN architecture comprises convolutional layers followed by max-pooling layers, flattening, and dense layers for classification.

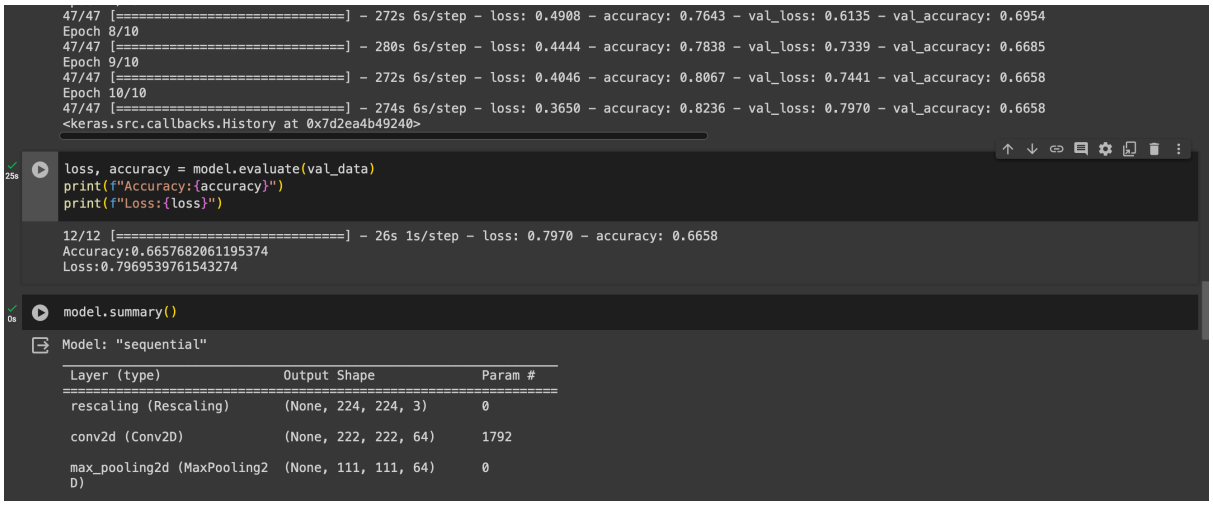
First normalization: Rescales input pixel values to the range [0, 1].

Then for the convolution layer:

- Filters/Neurons: 64 then 32 and 16
- Kernel Size: 3x3 (3 height, 3 width)
- Activation Function: ReLU (Rectified Linear Unit)

Finally, it flattens the output for input into the dense layers. And we have 2 dense layers, the last one using SoftMax activation for classification (only 2 choices in this case). I could have used Sigmoid but I'm just more confident with it, that's a personal preference.

I used Adam optimizer used for model compilation, it's the most used in our case and we will see later that it's the best.



As we can see the accuracy is bad for this model and the loss is growing fast, meaning that there are chances that we overfit.

I wanted to try a pretrained model to compare my result, just to see if it was really hard to get a good accuracy on this task or if my model needs improvements.

PRETRAINED RESNET50

ResNet50 is a convolutional neural network architecture comprised of 50 layers, utilizing skip connections to efficiently train deeper networks, enabling better performance in image recognition tasks.

I choosed it because it's one the most simple et precise model that I found. By the way, the 100 layers version exist but this one is enough for this case.

```python
# Import base model with Keras and freeze it
base_model = tf.keras.applications.resnet.ResNet50(include_top=False)
base_model.trainable = False

# Create inputs into the base model
inputs = tf.keras.layers.Input(shape=(224, 224, 3), name="input_layer")
x = base_model(inputs)

# Average pool the outputs of the base model (aggregate all the most important information, reduce number of computations)
x = tf.keras.layers.GlobalAveragePooling2D(name="global_average_pooling_layer")(x)

x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dense(512, activation="relu")(x)

# Create the output activation layer
outputs=tf.keras.layers.Dense(1, activation="sigmoid",name="output_layer")(x)

# Combine the inputs with the outputs into a model + compile
model= tf.keras.Model(inputs, outputs, name="model")

model.compile(loss=tf.keras.losses.BinaryCrossentropy(), # different loss function for Binary classifcation
              optimizer=tf.keras.optimizers.Adam(lr=0.001),
              metrics=["accuracy"]
             )

# Callbacks (stop training automatically once the model performance stop improving)
learning_rate_reduction = tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",patience=2,factor=0.5, min_lr=0.00001,verbose =1
Early_Stopping = tf.keras.callbacks.EarlyStopping(monitor="val_loss",patience=5, restore_best_weights=True)

# Fit the model
model.fit(train_data, epochs=5, validation_data=val_data, callbacks=[Early_Stopping,learning_rate_reduction], verbose=1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kern
94765736/94765736 [==============================] - 0s 0us/step
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate` or use the legacy optimizer, e.g.,tf.keras.optimize
Epoch 1/5
47/47 [==============================] - 170s 3s/step - loss: 0.5418 - accuracy: 0.8165 - val_loss: 0.1909 - val_accuracy: 0.9249
Epoch 2/5
47/47 [==============================] - 74s 1s/step - loss: 0.1327 - accuracy: 0.9524 - val_loss: 0.1617 - val_accuracy: 0.9303 -
Epoch 3/5
47/47 [==============================] - 72s 1s/step - loss: 0.0744 - accuracy: 0.9772 - val_loss: 0.1595 - val_accuracy: 0.9383 -
Epoch 4/5
47/47 [==============================] - 72s 1s/step - loss: 0.0550 - accuracy: 0.9812 - val_loss: 0.1108 - val_accuracy: 0.9571 -
Epoch 5/5
47/47 [==============================] - 67s 1s/step - loss: 0.0252 - accuracy: 0.9967 - val_loss: 0.1045 - val_accuracy: 0.9598 -
```

First, we import the model with the Keras API and we define the base model, it establishes an input layer of size (224, 224, 3)

Now the goal is to fine tune according to our needs using transfer learning. It is used to leverage pre-trained ResNet50 features for effective classification without retraining the entire model from scratch.

Pooling, Flatten and Dense (512 neurons) layers have been rewritten. Then I implemented callbacks including early stopping to avoid overfitting and learning rate reduction if validation loss plateaus.

Everything goes into a model, and it works exactly like before.

As you see the ResNet50 model goes up to 98-99% accuracy with nice loss values, our goal will be something close to it using this kind of callbacks and other techniques.

ENHANCED CNN

```
num_classes = 2

model = tf.keras.Sequential([
    layers.experimental.preprocessing.Rescaling(1./255), # Normalization pixel value from 0 to 1

    layers.Conv2D(256,5, activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(),

    layers.Conv2D(128,4, activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(),

    layers.Conv2D(64,4, activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(),

    layers.Conv2D(32,4, activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(),

    layers.Conv2D(16,4, activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(),

    layers.Flatten(),
    layers.Dense(128,activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(num_classes, activation='softmax')
])

optimizer = Adam()

def get_lr_metric(optimizer):
    def lr(y_true, y_pred):
        return optimizer._decayed_lr(tf.float32)
    return lr

lr_metric = get_lr_metric(optimizer)

model.compile(optimizer=optimizer, loss=tf.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['accuracy', lr_metric])
```

```
# Callbacks (TensorBoard, early stopping to prevent overfitting and lr reduction)
logdir="/content/drive/MyDrive/Tyre Quality Classification/logs"

learning_rate_reduction = tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",patience=3,factor=0.5, min_lr=0.00001,verbose=1)
early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)

history = model.fit(train_data,
          validation_data=val_data,
          epochs=40,
          callbacks=[early_stopping, learning_rate_reduction]
)
```

Back to our hand-made CNN, sequential is still used this is the same logic as before.

Things that I tested to get a better accuracy while avoiding overfitting:

- Increase the number of epochs, seeing that 30-50 is a good compromise between result and time of computation.

- Increase model depth and width by adding new Conv2D and Pooling2D layers with new and higher kernel size.

- Adding a dropout layer for regularization (avoid overfitting by disabling some neurons in the networks, O.5 = 50% of chance for a neuron to be disabled).

- Adding early stopping callback improved to avoid overfitting (based on loss value).

- Tried other optimizers (such as SGD & RMSprop) but Adam still the best (see tests in optimizers_test.ipynb).

- Adding adaptive learning rate callback, it adjusts the learning rate according to the loss value (see more details below).

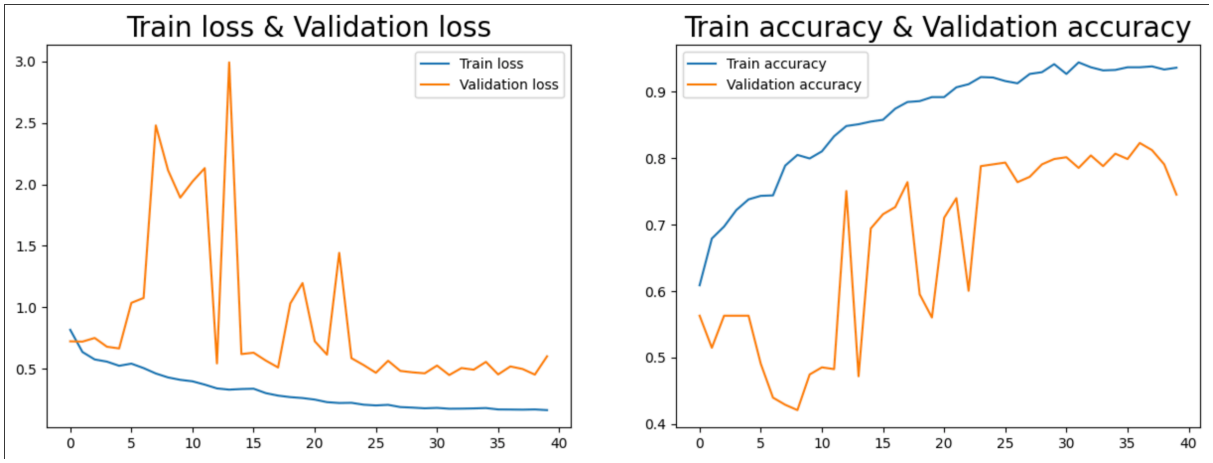
- Changed kernel size 3 -> 4, I noticed some improvements and tried to change input image size but it seems that 224 is the best for our task.

- Adding Batch Normalization to normalize the layers" inputs and acting as a form of regularization. Batch Normalization layers after each convolutional layer helps in stabilizing and normalizing the activations at each stage of the convolution process. This can lead to more stable and efficient training, potentially resulting in better performance and faster convergence.

For callbacks, I used EarlyStopping to avoid overfitting, ReduceLROnPlateau for learning rate reduction.
Early stopping has a patience of 10, fist I put 3 and I noticed it stop the training too early. The neural network can recover by itself if there are only 3 bad val_loss in a row, and it leads to much better result. Maybe 7-8 is a good value.

With all that new parameters we are now able to hits more than 92% of accuracy, 80% of val_accuracy with loss values that decrease. Not like in my first model which is great and proves that there is no overfitting and that there are fewer errors in the model.

```
Epoch 35: ReduceLROnPlateau reducing learning rate to 1e-05.
47/47 [==============================] - 72s 1s/step - loss: 0.1796 - accuracy: 0.9330 - lr: 1.5625e-05 - val_loss: 0.5550 - val_accuracy: 0.8070
Epoch 36/40
47/47 [==============================] - 64s 1s/step - loss: 0.1683 - accuracy: 0.9370 - lr: 1.0000e-05 - val_loss: 0.4530 - val_accuracy: 0.7989
Epoch 37/40
47/47 [==============================] - 71s 1s/step - loss: 0.1675 - accuracy: 0.9370 - lr: 1.0000e-05 - val_loss: 0.5180 - val_accuracy: 0.8231
```



Train values are logical and good but to be honest validation values are a bit spiky. In general accuracy increase and the loss decrease and stabilize that's what I wanted. Much better than the first CNN.

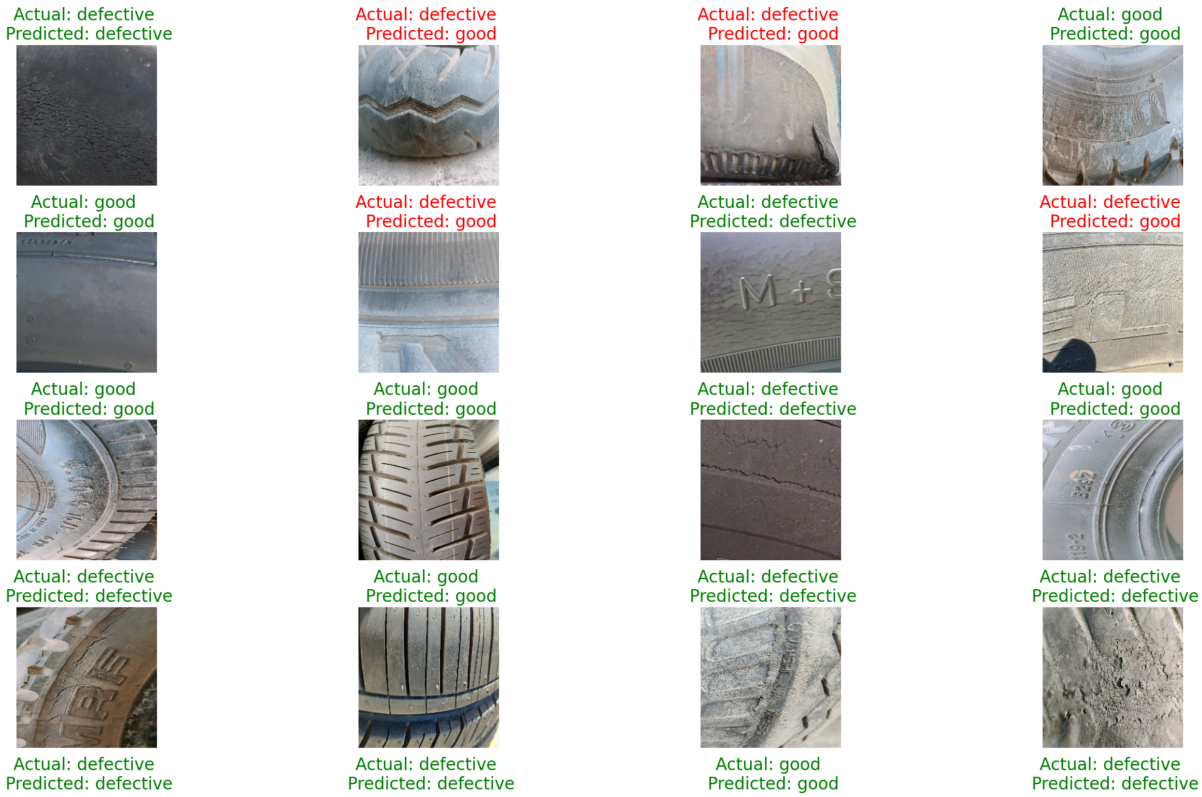
PREDICTIONS

The trained model is used to make predictions on the validation data (“x_val”) to classify the images. “y_val” for the labels.

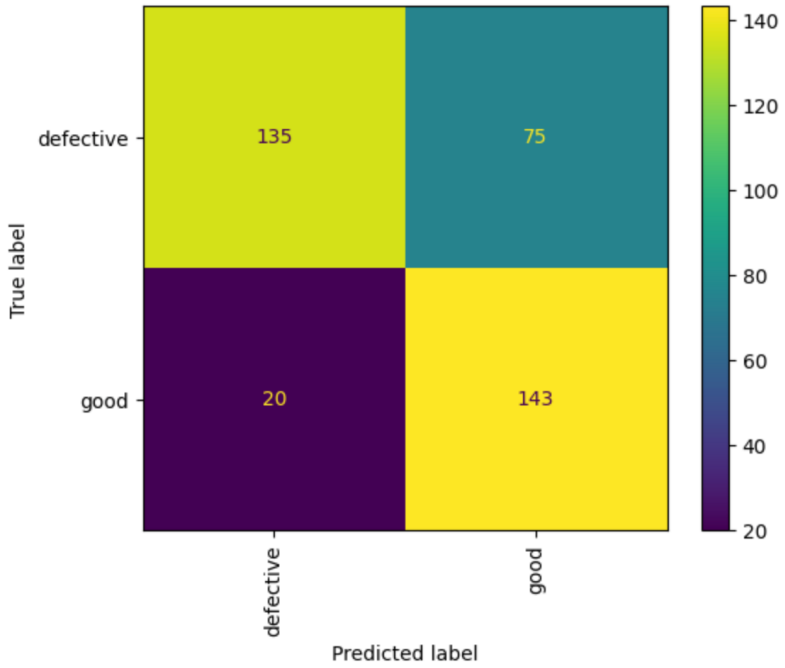
Of course, predictions are made on known data (val_data), in real life that should not be the case, data should not be known by the model. I was just lazy to find 10+ images like this on Google. The goal is just to show that it works.

The « model.predict » function generates predictions based on the trained model, which uses a SoftMax activation function to produce probabilities for each class.

So what I do is essentially going into each and every prediction and choosing the class with the highest probability using “np.argmax”. Here are the results of the predictions, green when the prediction is correct, red otherwise. So, we can see that the results are pretty good, the images are difficult to catch even for human.



Then I generated a confusion matrix just to visualize our result and the repartition. The brighter it is the better. We can see that in general the model is right. The model tends to be too kind and declare defective tires in good condition.



These are some additional calculations for my model, could be better if you stop at epoch 37.

My enhanced CNN model	
Recall score	0.763466
Precision score	0.760079
F1 score	0.745191

CONCLUSION

This project was a good practice to know more about CNN) and all the layers in it. I just made the connection between the theory (the course) and the practice. Even if we don't reach the accuracy of the pretrained model we obtain good results at the end.

The diversity of images in the dataset makes the problem even more challenging. In fact, the photos are taken from very different angles, sometimes up close, other times from afar. And we can see that the model has difficulty distinguishing between tires in good condition and tires in poor condition in the prediction results when the images are not explicit / clear.

That's what makes the difference between a good model and one where time has been taken to adjust each parameter.

BIBLIOGRAPHY

The lectures

A Comprehensive Guide to Convolutional Neural Networks - https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional Neural Network (CNN) - https://www.tensorflow.org/tutorials/images/cnn

Keras for Beginners: Implementing a Convolutional Neural Network - https://victorzhou.com/blog/keras-cnn-tutorial/

Kernel size and accuracy - https://datascience.stackexchange.com/questions/52889/does-increasing-kernel-size-in-a-cnn-result-in-higher-accuracy-on-the-training-s

Image resolution and accuracy - https://datascience.stackexchange.com/questions/84664/role-of-image-resolution-in-deep-learning

Basics of CNN - https://www.analyticsvidhya.com/blog/2022/03/basics-of-cnn-in-deep-learning/

Batch Normalization - https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization & https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/

Callbacks - https://www.tensorflow.org/api_docs/python/tf/keras/callbacks

Sequential - https://www.tensorflow.org/api_docs/python/tf/keras/Sequential